

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

11046 U.S. PTO
09/814324
03/21/01

Applicants: Kawahito et al
Serial No.:
Filed: herewith

Docket No. JP919990309US1
Art Unit: n/a
Dated:

For: PROGRAM COMPILATION AND OPTIMIZATION

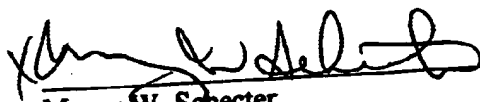
Assistant Commissioner for Patents
Washington, D.C. 20231

CLAIM OF PRIORITY

Sir:

Applicants, in the above-identified application, hereby claim the right of priority in connection with Title 35 U.S.C. &119 and in support thereof, herewith submits a certified copy of Japanese Patent Application No. 2000-114193, filed on April 14, 2000.

Respectfully submitted,


Manny W. Schecter
Reg. No. 31,722
Tel. (914) 945-3252

IBM CORPORATION
INTELLECTUAL PROPERTY LAW DEPT.
P.O. BOX 218
YORKTOWN HEIGHTS, NEW YORK 10598

ExpressMail EL715831397US
Date of Deposit: March 21, 2001

JP919990309

日 本 国 特 許 庁

PATENT OFFICE
JAPANESE GOVERNMENT

別紙添付の書類に記載されている事項は下記の出願書類に記載されて
る事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed
in this Office.

出 願 年 月 日
Date of Application:

2000年 4月14日

願 番 号
Application Number:

特願2000-114193

願 人
Applicant(s):

インターナショナル・ビジネス・マシーンズ・コーポレーシ
ョン

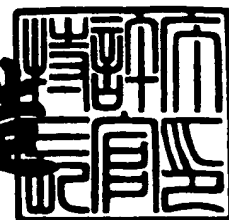
J1046 U.S. PRO
09/814324
03/21/01

CERTIFIED COPY OF
PRIORITY DOCUMENT

2000年12月22日

特許庁長官
Commissioner,
Patent Office

及 川 耕 造



出証番号 出証特2000-3107258

【書類名】 特許願

【整理番号】 JA999309

【提出日】 平成12年 4月14日

【あて先】 特許庁長官 殿

【国際特許分類】 G06F 5/06

【発明者】

 【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

 【氏名】 川人 基弘

【発明者】

 【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

 【氏名】 小笠原 武史

【発明者】

 【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

 【氏名】 小松 秀昭

【特許出願人】

 【識別番号】 390009531

 【氏名又は名称】 インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】

 【識別番号】 100086243

 【弁理士】

 【氏名又は名称】 坂口 博

【復代理人】

 【識別番号】 100104880

 【弁理士】

 【氏名又は名称】 古部 次郎

【選任した代理人】

【識別番号】 100091568

【弁理士】

【氏名又は名称】 市位 嘉宏

【選任した復代理人】

【識別番号】 100100077

【弁理士】

【氏名又は名称】 大場 充

【手数料の表示】

【予納台帳番号】 081504

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【包括委任状番号】 9706050

【包括委任状番号】 9704733

【プルーフの要否】 要

【書類名】 明細書

【発明の名称】 コンパイラ、コンピュータシステム、最適化方法、コンピュータプログラム、記憶媒体及びプログラム伝送装置

【特許請求の範囲】

【請求項 1】 プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラにおいて、

機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部と、

前記オブジェクトプログラムに対して、当該オブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点と当該例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行うプログラム変形部とを備えることを特徴とするコンパイラ。

【請求項 2】 前記プログラム変形部は、前記オブジェクトプログラム中の例外処理が起きる可能性のある命令に関して、前記例外処理の発生点と前記例外処理を実行する場所との間に内容的な相違がある場合に、当該内容的な相違を補償する補償コードを生成し、前記オブジェクトプログラムに挿入することを特徴とする請求項 1 に記載のコンパイラ。

【請求項 3】 前記プログラム変形部は、

前記最適化処理実行部による最適化処理に先立って、前記オブジェクトプログラム中の例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを設定する前処理部と、

前記最適化処理実行部により最適化処理を施された前記オブジェクトプログラムにおける前記例外処理が起きる可能性のある命令に対して、当該例外処理が起きる可能性のある命令と当該例外処理を実行する場所との間に内容的な相違があるか否かを調べ、相違がある場合に、当該内容的な相違を補償する補償コードと、当該補償コードを経た後に当該例外処理を実行する場所に制御を移すためのコードとを、前記キャッチ節の中に生成する後処理部とからなることを特徴とする請求項 1 に記載のコンパイラ。

【請求項 4】 前記プログラム変形部は、

前記最適化処理実行部による最適化処理に先立って、前記オブジェクトプログラム中の例外処理が起きる可能性のある命令を例外処理が起きるかどうかをチェックする命令と実際に例外処理を起こす命令とに分割すると共に、当該例外処理が起きた場合は当該実際に例外処理を起こす命令に制御を移すように、前記オブジェクトプログラムを変形することを特徴とする請求項 1 に記載のコンパイラ。

【請求項 5】 前記プログラム変形部は、

前記オブジェクトプログラム中の例外処理が起きる可能性のある命令が、例外処理が起きるかどうかをチェックする部分と実際に例外処理を起こす部分とで構成されている場合は、前記最適化処理実行部による最適化処理に先立って、当該命令を例外処理が起きるかどうかをチェックする命令と実際に例外処理を起こす命令とに分割すると共に、当該例外処理が起きた場合は当該実際に例外処理を起こす命令に制御を移すように、前記オブジェクトプログラムを変形し、

その他の場合は、前記最適化処理実行部による最適化処理に先立って、前記例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを設定し、

前記最適化処理実行部により最適化処理を施された前記オブジェクトプログラムにおける前記例外処理が起きる可能性のある命令に対して、当該例外処理が起きる可能性のある命令と当該例外処理を実行する場所との間に内容的な相違があるか否かを調べ、相違がある場合に、当該内容的な相違を補償する補償コードと、当該補償コードを経た後に当該例外処理を実行する場所に制御を移すためのコードとを、前記キャッチ節の中に生成することを特徴とする請求項 1 に記載のコンパイラ。

【請求項 6】 プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラを備えたコンピュータシステムにおいて、

前記コンパイラは、

機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部と、

前記オブジェクトプログラムに対して、当該オブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点と当該例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行うプログラム変形部とを備えることを特徴とするコンピュータシステム。

【請求項 7】 プログラムの処理効率を向上させるための最適化を行う最適化方法において、

処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、当該例外処理が起きるかどうかをチェックする部分を含み、かつ当該例外が起きた場合に当該例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成し、

前記例外処理の発生点と前記例外処理を実行する部分との間に内容的な相違がある場合に、当該内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成することを特徴とする最適化方法。

【請求項 8】 プログラムの処理効率を向上させるための最適化を行う最適化方法において、

処理対象であるプログラム中の例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを含むベーシックブロックを設定するステップと、

前記ベーシックブロックが設定された前記プログラムに対して最適化処理を実行するステップと、

前記最適化処理の施された前記プログラムにおいて、例外処理が起きる可能性のある命令と当該例外処理を実行する場所との間において、内容における相違があるか否かをチェックするステップと、

前記内容における相違がある場合に、前記ベーシックブロックの前記キャッチ節内に、当該相違を補償する補償コードと、補償コードを経た後に前記例外処理を実行する場所に制御を移すためのコードとを生成するステップとを含むことを特徴とする最適化方法。

【請求項 9】 前記例外処理が起きる可能性のある命令と当該例外処理を実

行する場所との間における内容的な相違をチェックするステップにおいて、内容が相違しない場合に、当該例外処理が起きる可能性のある命令に対する前記ベーシックブロックを除去するステップをさらに含むことを特徴とする請求項 8 に記載の最適化方法。

【請求項 1 0】 プログラムの処理効率を向上させるための最適化を行う最適化方法において、

処理対象であるプログラム中の例外処理が起きる可能性のあるコードを、例外処理が起きるかどうかをチェックするコードと、実際に例外処理を起こすコードとに分割するステップと、

前記コードを分割するステップで分割されたコードをコントロールフローグラフの分岐として明示するステップと、

例外処理が起きた場合に、前記実際に例外処理を起こすコードに制御を移すように前記コントロールフローグラフを構成するステップと、

変形の行われた前記プログラムに対して最適化処理を実行するステップとを含むことを特徴とする最適化方法。

【請求項 1 1】 前記最適化処理を実行した後、前記実際に例外処理を起こすコードを含むブロックに、例外処理の発生点と前記実際に例外処理を起こすコードとの間における内容的な相違を補償するコードが生成されているかどうかをチェックするステップと、

前記内容的な相違を補償するコードが生成されていない場合に、前記分割されたコード及びコントロールフローグラフを合成して分割前の状態に戻すステップとをさらに含むことを特徴とする請求項 1 0 に記載の最適化方法。

【請求項 1 2】 コンピュータにオブジェクトプログラムの最適化処理を実行させるコンピュータプログラムにおいて、

処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、当該例外処理が起きるかどうかをチェックする部分を含み、かつ当該例外処理が起きた場合に当該例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、

前記例外処理の発生点と前記例外処理を実行する部分との間に内容的な相違が

ある場合に、当該内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成する処理とを前記コンピュータに実行させることを特徴とするコンピュータプログラム。

【請求項 13】 コンピュータに実行させるコンピュータプログラムにおいて、

実行時に、前記コンピュータプログラム中の例外処理が起きる可能性のある命令に対して、

前記例外処理が起きるかどうかをチェックする機能と、

前記例外処理を実行する機能と、

前記例外処理が起きた場合に前記例外処理を実行する部分に制御を移すと共に、前記例外処理の発生点と前記例外処理を実行する部分との間に内容における相違がある場合に、前記例外処理を実行する部分への移行に先立って当該内容における相違を補償する機能とを備えることを特徴とするコンピュータプログラム。

【請求項 14】 前記例外処理が起きるかどうかチェックする機能は、トライキャッチブロックのトライ節または条件分岐であることを特徴とする請求項 13 に記載のコンピュータプログラム。

【請求項 15】 コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能に記憶した記憶媒体において、

前記プログラムは、

処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、当該例外処理が起きるかどうかをチェックする部分を含み、かつ当該例外処理が起きた場合に当該例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、

前記例外処理の発生点と前記例外処理を実行する部分との間に内容的な相違がある場合に、当該内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成する処理とを前記コンピュータに実行させることを特徴とする記憶媒体。

【請求項 16】 コンピュータに、
処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、当該

例外処理が起きるかどうかをチェックする部分を含み、かつ当該例外処理が起きた場合に当該例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、

前記例外処理の発生点と前記例外処理を実行する部分との間に内容的な相違がある場合に、当該内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成する処理とを実行させるプログラムを記憶する記憶手段と、

前記記憶手段から前記プログラムを読み出して当該プログラムを送信する送信手段とを備えたことを特徴とするプログラム伝送装置。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】

本発明は、例外処理が起きる可能性のある命令を含むプログラムに対して効果的な最適化処理を行うことが可能なコンパイラ及び最適化方法に関する。

【0002】

【従来の技術】

通常、プログラミング言語で記述されたプログラムのソース・コードをコンパイルする際、ソースプログラムから変換された機械語のオブジェクトプログラムに対して、実際の処理効率を向上させるための最適化処理を行っている。

【0003】

ところで、今日のプログラミング言語の多くは、仕様において、ユーザーが定義したエクセプション・ハンドラがある場合には、例外処理（エクセプション）が起きたときに、そのエクセプション・ハンドラに処理を移すことを規定している。また、例外処理が起きる可能性のある命令の数も多くなっている。

【0004】

このようなプログラミング言語においては、例外処理が起きた場合、それまでの実行状況を保証する必要がある。すなわち、例外処理が起きる可能性自体は低い、これを無視することはできないため、例外処理が起きた場合に対応するための処理を用意しておかなくてはならない。しかし、例外処理が起きる可能性が低いということは、そのための対応処理が必要となることも少ないため、全体と

して動作効率が落ちることとなる。そのため、プログラム中に例外処理が起きる可能性のある命令がある場合とそのような命令が無い場合とを比べると、上述したプログラムのコンパイル時における最適化の効果が低下していた。

【 0 0 0 5 】

また従来、エクセプション・ハンドラを含む関数については、トライ領域 (Try Region) 内で通常のレジスタ割付を行うと、例外処理が起きた場合に、変数の値がどこに格納されているかを特定することができなかった。このため、エクセプション・ハンドラ以降で参照がある変数については、例外処理が起きたか否かに関わらず、基本的にメモリ上にあるものとして扱っていた。したがって、例外処理が起きない場合でも、当該変数をメモリ上に読みに行かなければならないため、全体的な実行速度を低下させることとなっていた。

【 0 0 0 6 】

このようなプログラムの最適化に対する弊害を防止するため、プログラムにおいて例外処理が起きる可能性のある命令を減らすような技術が、従来から提案されている。そのような従来技術としては、例えば、ループのバージョニングやデータフローアナリシスを用いたエクセプション・チェックの除去等がある。

これらの従来技術によれば、例外処理が起きる可能性のある中間コードそのものがなくなるため、エクセプション・チェックのコストを削減することができ、それだけでなく各種の最適化処理の効果を高めることができる。

【 0 0 0 7 】

【発明が解決しようとする課題】

上述したように、例外処理に対してエクセプション・ハンドラを設定して処理を移行するプログラミング言語にて記述されたプログラムは、プログラム中に例外処理が起きる可能性のある命令を含む場合、最適化の効率が悪かった。

【 0 0 0 8 】

これに対し、従来から提案されているプログラムにおいて例外処理が起きるがある命令を減らす手法により、例外処理が起きる可能性のある中間コードを減らし、最適化の効果を高めることが可能である。

しかし、この従来の手法は、全ての例外処理が起きる可能性のある中間コード

を除去できるわけではなく、例えば、関数呼び出しの命令等は除去できなかった。そのため、除去できない命令に関しては、依然として最適化の効果を低下させる原因となっていた。

【0009】

そこで本発明は、例外処理が起きる可能性のある命令を含むプログラムに対して効果の高い最適化を行うことを目的とする。

【0010】

【課題を解決するための手段】

かかる目的のもと、本発明は、プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラにおいて、機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部と、このオブジェクトプログラムに対して、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点とこの例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行うプログラム変形部とを備えることを特徴としている。

これにより、例外処理が起きる可能性のある命令において、例外処理が起きた場合を考慮して特別に扱うこと無く、通常最適化処理を実行することが可能となる。

【0011】

ここで、このプログラム変形部は、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に関して、この例外処理の発生点とこの例外処理を実行する場所との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードを生成し、このオブジェクトプログラムに挿入することを特徴としている。

例外処理の発生点とこの例外処理を実行する場所との間における内容的な相違とは、例えばレジスタ割付やメモリ内容的な相違である。これらの具体的な相違の内容に応じて、適切な補償コードを挿入することとなる。

【0012】

またここで、このプログラム変形部は、最適化処理実行部による最適化処理に

先立って、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを設定する前処理部と、最適化処理実行部により最適化処理を施されたオブジェクトプログラムにおける例外処理が起きる可能性のある命令に対して、この例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間に内容的な相違があるか否かを調べ、相違がある場合に、この内容的な相違を補償する補償コードと、この補償コードを経た後にこの例外処理を実行する場所に制御を移すためのコードとを、キャッチ節の中に生成する後処理部とからなることを特徴としている。

例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間に内容的な相違がない場合には、前処理部で作成したトライ節とキャッチ節とを削除することができる。

【 0 0 1 3 】

さらにここで、このプログラム変形部は、前記最適化処理実行部による最適化処理に先立って、このオブジェクトプログラム中の例外処理が起きる可能性のある命令を例外処理が起きるかどうかをチェックする命令と実際に例外処理を起こす命令とに分割すると共に、この例外処理が起きた場合はこの実際に例外処理を起こす命令に制御を移すように、このオブジェクトプログラムを変形することを特徴としている。

このように変形したオブジェクトプログラムに対して最適化処理を行った後に、実際に例外処理を起こす命令を含むブロックにおいて、例外処理の発生点とこの例外処理を実行する場所との間における内容的な相違を補償するためのコードが生成されていなければ、分割した命令を合成し、分割前の命令に戻すことができる。

【 0 0 1 4 】

さらにここで、このプログラム変形部は、このオブジェクトプログラム中の例外処理が起きる可能性のある命令が、例外処理が起きるかどうかをチェックする部分と実際に例外処理を起こす部分とで構成されている場合は、最適化処理実行部による最適化処理に先立って、この命令を例外処理が起きるかどうかをチェッ

クする命令と実際に例外処理を起こす命令とに分割すると共に、この例外処理が起きた場合はこの実際に例外処理を起こす命令に制御を移すように、このオブジェクトプログラムを変形し、その他の場合は、最適化処理実行部による最適化処理に先立って、この例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを設定し、かつ最適化処理実行部により最適化処理を施されたこのオブジェクトプログラムにおけるこの例外処理が起きる可能性のある命令に対して、この例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間に内容的な相違があるか否かを調べ、相違がある場合に、この内容的な相違を補償する補償コードと、この補償コードを経た後にこの例外処理を実行する場所に制御を移すためのコードとを、キャッチ節の中に生成することを特徴としている。

【 0 0 1 5 】

また、本発明は、プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラを備えたコンピュータシステムにおいて、このコンパイラは、機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部と、このオブジェクトプログラムに対して、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点とこの例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行うプログラム変形部とを備えることを特徴としている。

このような構成とすれば、このコンピュータシステムにおいて、プログラムをコンパイルする場合に、例外処理が起きる可能性のある命令において、例外処理が起きた場合を考慮して特別に扱うこと無く、通常の最適化処理を実行することが可能となる点で好ましい。

【 0 0 1 6 】

さらにまた、本発明は、プログラムの処理効率を向上させるための最適化を行う最適化方法において、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、この例外処理が起きるかどうかをチェックする部分を含み、かつこの例外処理が起きた場合にこの例外処理を実行する部分に制御を移行させ

る命令を持つベーシックブロックを作成し、この例外処理の発生点とこの例外処理を実行する部分との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成することを特徴としている。

これにより、例外処理が起きる可能性のある命令において、例外処理が起きた場合を考慮して特別に扱うこと無く、通常の最適化処理を実行することが可能となる。

【 0 0 1 7 】

さらに、本発明は、プログラムの処理効率を向上させるための最適化を行う最適化方法において、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを含むベーシックブロックを設定するステップと、このベーシックブロックが設定されたプログラムに対して最適化処理を実行するステップと、この最適化処理の施されたプログラムにおいて、例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間において、内容における相違があるか否かをチェックするステップと、この内容における相違がある場合に、このベーシックブロックのキャッチ節内に、この相違を補償する補償コードと、補償コードを経た後にこの例外処理を実行する場所に制御を移すためのコードとを生成するステップとを含むことを特徴としている。

最適化処理の前にベーシックブロックを設定しておき、最適化の後に、補償の必要性をチェックして補償コードを生成することにより、例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間における内容的な相違を適切に補償することができる。

【 0 0 1 8 】

ここで、さらに、この例外処理が起きる可能性のある命令と当該例外処理を実行する場所との間における内容的な相違をチェックするステップにおいて、内容が相違しない場合に、当該例外処理が起きる可能性のある命令に対する前記ベーシックブロックを除去するステップを含むこととすることができる。

不要なベーシックブロックを除去することにより、最適化後のオブジェクトプログラムのコード量を削減することができる。

【 0 0 1 9 】

また、本発明は、プログラムの処理効率を向上させるための最適化を行う最適化方法において、処理対象であるプログラム中の例外処理が起きる可能性のあるコードを、例外処理が起きるかどうかをチェックするコードと、実際に例外処理を起こすコードとに分割するステップと、コードを分割するステップで分割されたコードをコントロールフローグラフの分岐として明示するステップと、例外処理が起きた場合に、この実際に例外処理を起こすコードに制御を移すようにこのコントロールフローグラフを構成するステップと、この変形が行われたプログラムに対して最適化処理を実行するステップとを含むことを特徴としている。

このように変形されたプログラムは、例外処理を起こすコードにおいて、必要に応じて、例外処理の発生点と例外処理を実行する場所との間における内容的な相違を補償するコードが含まれている。したがって、この変形の後にこのプログラムに対して最適化処理を行うことにより、効率的な最適化が実現される。

【 0 0 2 0 】

ここで、さらに、最適化処理を実行した後、実際に例外処理を起こすコードを含むブロックに、例外処理の発生点とこの実際に例外処理を起こすコードとの間における内容的な相違を補償するコードが生成されているかどうかをチェックするステップと、この内容的な相違を補償するコードが生成されていない場合に、この分割されたコード及びコントロールフローグラフを合成して分割前の状態に戻すステップとを含むこととすることができる。

無意味な分岐を元の状態に戻すことにより、最適化後のオブジェクトプログラムのコード量を削減することができる。

【 0 0 2 1 】

また、本発明は、コンピュータにオブジェクトプログラムの最適化処理を実行させるコンピュータプログラムにおいて、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、この例外処理が起きるかどうかをチェックする部分を含み、かつこの例外処理が起きた場合にこの例外処理を実行する部分

に制御を移行させる命令を持つベーシックブロックを作成する処理と、この例外処理の発生点とこの例外処理を実行する部分との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードをこのベーシックブロックの中に生成する処理とをコンピュータに実行させることを特徴としている。

このような構成とすることにより、このプログラムをインストールしたコンピュータにおいて、例外処理が起きる可能性のある命令を含むプログラムに対して効果的な最適化を行うことが可能となる。

【 0 0 2 2 】

また、本発明は、コンピュータに実行させるコンピュータプログラムにおいて、実行時に、このコンピュータプログラム中の例外処理が起きる可能性のある命令に対して、この例外処理が起きるかどうかをチェックする機能と、この例外処理を実行する機能と、この例外処理が起きた場合にこの例外処理を実行する部分に制御を移すと共に、この例外処理の発生点とこの例外処理を実行する部分との間に内容における相違がある場合に、この例外処理を実行する部分への移行に先立ってこの内容における相違を補償する機能とを備えることを特徴としている。

このように構成されたプログラムは、例外処理が起きる可能性のある命令を含んでいても、このような命令を、例外処理を考慮して特別に扱うことなく最適化できるため、実行速度が向上する点で優れている。

【 0 0 2 3 】

ここで、例外処理が起きるかどうかチェックする機能は、トライーキャッチブロックのトライ節または条件分岐により実現することができる。

【 0 0 2 4 】

また、本発明は、コンピュータに実行させるプログラムをこのコンピュータの入力手段が読取可能に記憶した記憶媒体において、このプログラムは、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、この例外処理が起きるかどうかをチェックする部分を含み、かつこの例外処理が起きた場合にこの例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、この例外処理の発生点とこの例外処理を実行する部分との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードをこのベ

ーシックブロックの中に生成する処理とをこのコンピュータに実行させることを特徴としている。

このような構成とすることにより、このプログラムをインストールしたコンピュータにおいて、例外処理が起きる可能性のある命令を含むプログラムに対して効果的な最適化を行うことが可能となる。

【 0 0 2 5 】

コンピュータに、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、この例外処理が起きるかどうかをチェックする部分を含み、かつこの例外処理が起きた場合にこの例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、この例外処理の発生点とこの例外処理を実行する部分との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードをこのベーシックブロックの中に生成する処理とを実行させるプログラムを記憶する記憶手段と、この記憶手段からこのプログラムを読み出してこのプログラムを送信する送信手段とを備えたことを特徴としている。

このような構成とすることにより、このプログラムをダウンロードしたコンピュータにおいて、例外処理が起きる可能性のある命令を含むプログラムに対して効果的な最適化を行うことが可能となる。

【 0 0 2 6 】

【発明の実施の形態】

以下、添付図面に示す実施の形態に基づいてこの発明を詳細に説明する。

まず、本発明の概要を説明する。なお、以下では、オブジェクト指向プログラミング言語で記述されたプログラムを前提として説明する。しかし、本発明による最適化技術の適用範囲は、ユーザーが定義したエクセプション・ハンドラがある場合に、例外処理が起きたならばそのエクセプション・ハンドラに処理を移す仕様を持ったプログラミング全般に及び、オブジェクト指向プログラミング言語に限定されるものではない。

本発明の最適化方法は、例外処理が起きる可能性のある命令を含むプログラムにおいて、例外発生点とエクセプション・ハンドラ（メソッド外も含む）までの間に、最適化による2点の違い（例えばレジスタ割付やメモリ内容）を吸収する

ための補償コードを生成する。この変形により、当該プログラムの実行時において、例外発生時には、システムは例外発生点から補償コードの存在を確認し、存在するならば、当該エクセプション・ハンドラの代わりに、まず補償コード点へ飛び、その後エクセプション・ハンドラに制御を移す。

【 0 0 2 7 】

図 1 は、例外処理が起きる可能性のある命令において、エクセプション・ハンドラへ移行する様子を示す図である。図 1 において、例外処理 1 では、例外処理が発生しても補償コードを要することなくエクセプション・ハンドラへ移行する。これに対し、例外処理 2、3 では、例外発生点とエクセプション・ハンドラとの間の違いを吸収するために、補償コードを経てエクセプション・ハンドラへ移行する。このように、必要に応じて例外発生点から直接エクセプション・ハンドラへ移行するのではなく、補償コードを介することにより、例外処理 1、2、3 のいずれにおいても、エクセプション・ハンドラにおいては同じレジスタイメージを保証することができる。

【 0 0 2 8 】

これにより、例外処理が起きる可能性のある命令において、例外処理が発生した場合のことを考慮して、変数をメモリに読みに行くなどの特別な扱いを行う必要が無くなる。したがって、例外処理が起きる可能性のある命令を特別に扱わず最適化できるため、例外処理が発生しない場合の最適化の効率を向上させることができ、全体的な実行速度を向上させることができる。

【 0 0 2 9 】

補償コードの挿入は、プログラム中にベーシックブロックを作成し、当該ベーシックブロック中に当該補償コードを生成することにより行われる。このベーシックブロックによる補償コードを挿入する場所を作成する方法として、Try-Catch ブロックを作成する方法と、明示的に条件分岐を設定する方法とがある。以下、各方法を分けて説明する。

【 0 0 3 0 】

〔第 1 の実施の形態〕

図 2 は、第 1 の実施の形態によるコンパイラにおける最適化処理部の構成を説

明する図である。

図 2 において、符号 1 0 は最適化処理部であり、機械語のオブジェクトコード（中間コード）に変換されたプログラムを最適化する。符号 1 1 は前処理部であり、補償コードを生成するためのベーシックブロックを作成する。符号 1 2 は最適化処理実行部であり、前処理部 1 1 により前処理の施されたオブジェクトプログラムに対して通常の最適化処理を実行する。符号 1 3 は後処理部であり、前処理部 1 1 にて作成されたベーシックブロックの中に例外処理に対応するための補償コードを設定する。

【 0 0 3 1 】

本実施の形態において、前処理部 1 1 は、プログラム中の例外処理が起きる可能性のある命令に対して、ベーシックブロックとして Try-Catch ブロックを設定する。ここで、Try-Catch ブロックは、例外処理が起きる可能性のある命令を Try 節と Catch 節とで挟む。そして、Try 節で例外処理が起きたか否かをチェックし、例外処理が発生したならば、Catch 節の処理に移行する。したがって、この Catch 節の中に補償コードが生成されることとなる。

【 0 0 3 2 】

最適化処理実行部 1 2 は、通常の最適化処理を実行する。最適化処理の対象であるプログラムは、前処理部 1 1 により例外処理が起きる可能性のある命令に対して、Try-Catch ブロックを設けているため、全体としては例外処理を考慮して特別に扱う必要がない。したがって、効果の高い最適化を実現することができる。

【 0 0 3 3 】

後処理部 1 3 は、Try-Catch ブロックの Catch 節に必要な補償コードを生成する。また、補償コードを必要としない場合は、前処理部 1 1 にて作成された Try-Catch ブロックを除去する。ここで、補償コードを必要としない場合とは、例外処理が起きる可能性のある命令と、本来の当該例外処理を実行する場所であるエクセプション・ハンドラとの間に相違点がない場合である。この場合、図 1 に示した例外処理 1 のように、そのままエクセプション・ハンドラへ移行すればよい。したがって、補償コードは必要なく、本実施の形態による Try-Catch ブロ

ックも不要となるため、プログラムから除去する。

【 0 0 3 4 】

図 3 は、本実施の形態における最適化処理の概略的な流れを説明するフローチャートである。

図 3 を参照すると、オブジェクトコード（中間コード）に変換されたプログラムを入力した最適化処理部 1 0 において、まず、前処理部 1 1 が、メソッド内の例外処理が起きる可能性のある命令について補償コード生成のために Try-Catch ブロックを設定する（ステップ 3 0 1）。

次に、最適化処理実行部 1 2 が、最適化処理を実行する（ステップ 3 0 2）。

次に、後処理部 1 3 が、例外処理が起きる可能性がある命令とエクセプション・ハンドラとの間において、レジスタイメージやメモリ内容等に相違点があるか否かをチェックする（ステップ 3 0 3）。そして、相違点があるならば、前処理部 1 1 において作成した Try-Catch ブロックの C a t c h 節内に当該相違を補償する補償コードと、補償コードを経た後にエクセプション・ハンドラに制御を移すためのコードとを生成する（ステップ 3 0 4、3 0 5）。

また、例外処理が起きる可能性がある命令とエクセプション・ハンドラとの間に相違点がないならば、当該 Try-Catch ブロックを除去する（ステップ 3 0 3、3 0 6）。

【 0 0 3 5 】

本実施の形態によれば、例外処理が起きる可能性のある命令について、プログラムの実行時に例外処理が起きない場合は、本実施の形態を適用しない場合と比べて同等以上の実行速度が得られる。これに対し、実行時に例外処理が起きた場合は、本実施の形態を適用しない場合と比べて遅くなる。しかしながら、例外処理が起きる可能性のある命令について、例外処理が起きない頻度は、例外処理が起きる頻度をかなり上回ると予想され、無視することができる。

【 0 0 3 6 】

ここで、例外処理が発生した場合に補償コードを見つける手法について説明する。上述したように、本実施の形態では、補償コードを挿入するために Try-Catch ブロックを用いる。しかし、従来の Try-Catch の実現方法として、T r y 節一つ

に対してエクセプション・ハンドラを対応させる方法があった。この方法では、T r y 節の境界でT r y 情報を記憶しておき、例外処理の発生時には、記憶したT r y 情報に基づいてエクセプション・ハンドラを検出していた。

しかし、本実施の形態を適用すると、メソッド内の複数のエクセプションが起きる可能性がある命令に対して、補償コードを挿入するためのTry-Catchブロックが生成される。この時、従来のTry-Catchの実現方法を用いて補償コードを見つけるようにすると、T r y 節の境界でT r y 情報を記憶するために実行速度が遅くなってしまう。

そこで、本実施の形態では、Try-Catchの実現方法として例外発生点のアドレスと例外処理の種類に基づいて、対応するエクセプション・ハンドラを検出し、制御を移すような機構を導入する。この機構を導入することにより、実行速度が遅くなることを回避することができる。

【 0 0 3 7 】

次に、Try-Catchブロックにおいて例外発生点からエクセプション・ハンドラに制御を移すための手法について説明する。

まず、コンパイル時に、後処理部 1 3 が、トライ領域内の例外処理が起きる可能性のある命令について例外発生点のアドレス、対応するハンドラのアドレス、ハンドラのエクセプションの種類を登録する。その命令で起こった例外処理により制御が移る可能性があるエクセプション・ハンドラが複数ある時は、対応するハンドラごとにこれらの情報を全て登録する。

【 0 0 3 8 】

次に、プログラムの実行時に、以下の手順で、例外発生コードからエクセプション・ハンドラに制御を移すための処理を行う。

まず、メソッド内で例外処理が起きたときには、登録されている例外発生点のアドレスと例外処理の種類とを検索し、対応するエクセプション・ハンドラのアドレスを取得する。

そして、検索に成功したならば、検出したエクセプション・ハンドラに制御を移す。

一方、検索に失敗したならば、現メソッドを呼んだメソッドに制御を移し、検

索を続行する。

この方法を適用した場合、例外処理が起きない場合は、従来のトライ領域の境界で記録する方法と比べて高速に実行することが可能である。一方、登録テーブルを生成するため、従来と比べて、コンパイラが生成するコード量が増えるおそれがある。しかしながら、上述したように、後処理部 1 3 において、例外処理が起きる可能性がある命令とエクセプション・ハンドラとの間に相違点がない場合、すなわち最適化の効果が無かった変形は、元の形に戻すようにしている。このため、登録テーブルのサイズも最小限必要なものに限られている。したがって、コード量の増大も最小限に抑えることができる。

【 0 0 3 9 】

次に、本実施の形態を用いて例外処理に対応するための補償コードを挿入する具体的な実施例を説明する。

まず、エクセプション・ハンドラ以降で参照がある変数に関して、トライ領域内で通常のレジスタ割付を使って、当該変数をレジスタに割り付けることを可能にする方法について説明する。

【 0 0 4 0 】

本実施の形態による補償コードの挿入を、次のように適用することによって、エクセプション・ハンドラ以降で参照がある変数に関して、トライ領域内で通常のレジスタ割付を使って、当該変数をレジスタに割り付けることが可能となる。また、例外処理が起きた場合でも、コンパイラにより生成されたエクセプション・ハンドラ内の補償コードにより、正しく動作させることができる。

まず、エクセプション・ハンドラ以降で参照がある変数の集合情報をトライ領域ごとに求める。また、トライ領域内ではエクセプションが起きる命令についてエクセプション・ハンドラを設定する。

さらに、エクセプション・ハンドラ内において、最初に求めた集合情報に基づいて補正コードである調整コードを生成する。

そして、エクセプション・ハンドラの最後で、本来移行するはずだったエクセプション・ハンドラに制御を移す。

【 0 0 4 1 】

以上の動作を、図4乃至図7の擬似コードを参照しながらさらに詳細に説明する。なお、以下の説明において、変数の値設定をDEF、変数の使用をUSE、トライ領域の識別番号をTRYとする。

まず、個々のトライ領域に対応するエクセプション・ハンドラの開始点で、全てのローカル変数についてDEFがあると仮定し、そのDEFを参照するローカル変数の集合：USE_VIA_EH_LIST[TRY] を求める。この動作は、次の三つの動作からなる。なお、以下の説明及び図示のコードにおいて、reach[bidx].in、reach[bidx].out、reach[bidx].gen、reach[bidx].killは、トライ領域の識別番号およびローカル変数番号の2つを持つ要素からなる集合である。

【0042】

A. 図4に示すアルゴリズムを用い、各ベーシックブロックのreach[bidx].gen及びreach[bidx].killを求める。

B. A. で求めたgen及びkillを用い、図5に示す式でデータフローアナリシスを行い、エクセプション・ハンドラの開始点で設定したDEFが各ベーシックブロックの先頭に到達する集合reach[B].inを求める。

C. B. で求めたreach[B].inを用い、図6に示すアルゴリズムにより、エクセプション・ハンドラの開始点で設定したDEFを参照するローカル変数の集合USE_VIA_EH_LIST[TRY] を求める。

【0043】

次に、トライ領域内で対応するエクセプション・ハンドラに到達する可能性がある例外処理が起きる命令に対して、図7に示すアルゴリズムを用いて、新たなエクセプション・ハンドラを設定する。

図8は、図7のアルゴリズムにおける
 <この命令について新たなエクセプション・ハンドラを設定する。>
 の処理内容を説明するフローチャートである。

図8を参照すると、まず、USE_VIA_EH_LIST[TRY]内のローカル変数に対して、本来のエクセプション・ハンドラの入り口でのローカル変数のレジスタイメージを決める（ステップ801）。そして、当該レジスタイメージに合うように、メモリまたは特定のレジスタへローカル変数値をコピーするコードの集合情報を求

める（ステップ 8 0 2）。

【 0 0 4 4 】

次に、求めた集合情報が空集合かどうかを調べる（ステップ 8 0 3）。空集合でなければ、次に、新しいベーシックブロックを作成する（ステップ 8 0 4）。そして、作成したベーシックブロック内に、ステップ 8 0 2 で作成した集合情報のコードを生成する（ステップ 8 0 5）。

次に、当該ベーシックブロックに対して、ステップ 8 0 5 で生成したコードの後に、本来のエクセプション・ハンドラに制御を移すためのコードを生成する（ステップ 8 0 6）。

【 0 0 4 5 】

最後に、例外処理を起こす可能性がある命令に対して、Try-Catchブロックを作成し、C a t c h点として、ステップ 8 0 4 で作成したベーシックブロックを設定する（ステップ 8 0 7）。

なお、ステップ 8 0 3 において、ステップ 8 0 2 で求めた集合情報が空集合であったならば、ステップ 8 0 4 以降の動作を行わずに終了する。

以上の処理を行うことにより、エクセプション・ハンドラ以降で参照がある変数についても、トライ領域内で通常のレジスタ割付を使ってレジスタに割り付けることが可能になる。

【 0 0 4 6 】

次に、本実施の形態における補償コードの挿入によりプログラムの実行速度が速くなる具体例を示す。

図 9 は、配列内の最小値及び最大値を求める J a v a のサンプルプログラムを示す図である。また、図 1 0 は、図 9 のプログラムに対して本実施の形態により補償コードを挿入する変形を加えた状態を示す図である。

【 0 0 4 7 】

図 9 に示すプログラムにおいて、まず、エクセプション・ハンドラの開始点で設定した D E F を参照するローカル変数の集合：USE_VIA_EH_LIST[TRY] を求める

。

ローカル変数 i 及び j は、プログラム中の（1 6）行で U S E がある。また、

最小値 min 及び最大値 max は、(19) 行、(20) 行で USE があるが (17) 行、(18) 行において、エクセプション・ハンドラの開始点で設定した DEF は壊される。

また、 a 、 size_x 、 size_y に関しては、エクセプション・ハンドラ以降に USE はない。よって、 $\text{USE_VIA_EH_LIST}[\text{TRY}]$ は、 i 及び j となる。この2つの変数について、エクセプション・ハンドラの入り口でのレジスタイメージは、それぞれ $R1$ 、 $R2$ とする。

【0048】

次に、新たなエクセプション・ハンドラを設定する。

トライ領域内で、対応するエクセプション・ハンドラに到達する可能性がある例外処理が起きる命令は、(3) 行及び(9) 行である。これらの命令に対してエクセプション・ハンドラを設定すると、図10のように変形される。図10を参照すると、(3) 行は(3. 1) 行から(3. 4) 行まで、(9) 行は(9. 1) 行から(9. 3) 行までにそれぞれ示すように、命令を細かい単位に分けてある。なお、 NULLCHECK とは NULL かどうかをチェックする中間コードを表し、 SIZECHECK とは配列のサイズとインデックスとをチェックする中間コードを表している。

【0049】

図9のプログラムを図10に示すように変形することによって、トライ領域内における例外処理が起きないパスにおいて、全ての変数を通常のレジスタ割付を使ってレジスタに割り付けることが可能になる。

【0050】

次に、本実施の形態を用いて例外処理に対応するための補償コードを挿入する他の実施例として、部分的な冗長度除去 (Partial Redundancy Elimination) のアルゴリズムを使ったメモリへの書き込み命令の最適化の効果を高める方法について説明する。

【0051】

図11は、本実施例における動作を説明するフローチャートである。

図11を参照すると、まず、例外処理が起きる可能性のある中間コードに対し

、新たなエクセプション・ハンドラを設定する（ステップ 1 1 0 1）。当該エクセプション・ハンドラは、当該中間コードによって起き得る全てのエクセプションを `C a t c h` するように設定する。

【 0 0 5 2 】

例外処理が起きる可能性のある中間コードにおける当該例外処理が 1 種類であり、かつ対応するエクセプション・ハンドラが元々メソッド内にあるならば、`C a t c h` 節の中に、補償コードとして、当該エクセプション・ハンドラにジャンプするコードを生成する。それ以外の場合は、受け取った例外処理を `Throw` するコードを生成する（ステップ 1 1 0 2）。

【 0 0 5 3 】

また、上記の例外処理が起きる可能性のある中間コードの下でベーシックブロックを分割し、`C a t c h` 節のベーシックブロックと例外処理が起きる可能性のある中間コードに後続する中間コードとの二つの間にコントロールフローグラフのエッジを張る（ステップ 1 1 0 3）。

【 0 0 5 4 】

次に、ステップ 1 1 0 3 で作成されたコントロールフローグラフに対して、メモリへの書き込み命令の最適化を行う（ステップ 1 1 0 4）。

最後に、ステップ 1 1 0 1 で作成したエクセプション・ハンドラ に対して、`C a t c h` 節のベーシックブロックに、ステップ 1 1 0 2 で生成した `goto` 命令または `throw` 命令しか存在しなければ `Try-Catch` ブロックを取り外し、ステップ 1 1 0 1 乃至ステップ 1 1 0 4 による変形前の状態に戻す（ステップ 1 1 0 5）。

【 0 0 5 5 】

図 1 2 は例外処理が起きる可能性のある中間コードを含むプログラムのコントロールフローグラフ、図 1 3 は図 1 2 のコントロールフローグラフに対して図 1 1 に示した動作により補償コードである `goto` 命令または `Throw` 命令を挿入した状態を示すコントロールフローグラフを、それぞれ示す図である。

【 0 0 5 6 】

次に、本実施の形態におけるメモリへの書き込み命令の最適化の具体的な例を示す。

図 1 4 は、J a v a のサンプルプログラムを示す図である。また、図 1 5 は、図 1 4 のサンプルプログラムを細かい処理単位に分けたプログラムである。

以下の説明において、mem_で始まる変数はメモリ上にある変数を指し、reg_で始まる変数はレジスタ上にある変数を指すものとする。また、図 1 5 において、NULLCHECK reg_aとは、変数reg_aがnullかどうかをテストし、nullならば、例外処理であるNullPointerエクセプションを起こすような処理を表す。また、SIZECHECK reg_pos, reg_a[]とは、 $0 \leq \text{reg_pos} < \text{arraylength}(\text{reg_a})$ かどうかをテストし、条件に当てはまらなければ、例外処理であるArrayIndexOutOfBoundsエクセプションを起こすような処理を表している。

【 0 0 5 7 】

図 1 6 は、図 1 5 に示すプログラムに対して、従来技術による最適化処理を行った状態を示す図である。

図 1 6 において、斜体の文字で記述した部分 1 6 0 1、1 6 0 2 は、ループ内で実行される中間コードを表している。例外処理が起きる可能性がある中間コードがループ内に存在しているため、①の中間コードをループの外に出すことができない。

【 0 0 5 8 】

図 1 7 は、図 1 6 の状態のプログラムに対して、本実施の形態による補償コードの挿入を行った状態を示す図である。

図 1 7 において、ループ内の例外処理が起きる可能性がある中間コードに対して、Try-Catchブロックを作成し、C a t c h 節内で受け取ったエクセプションをthrowしている（1 7 0 1、1 7 0 2 参照）。

【 0 0 5 9 】

図 1 8 は、図 1 7 の状態のプログラムに対して、図 1 1 に示した部分的な冗長度除去（Partial Redundancy Elimination）のアルゴリズムを用いてメモリ書き込み命令の最適化を行った状態を示す図である。

図 1 8 において、斜体の文字で記述した部分 1 8 0 1、1 8 0 2、1 8 0 3 は、例外処理が起きないときにループ内で実行される中間コードを表している。例外処理が起きない場合のパスを考えると、図 1 6 と比べてメモリへの書き込みが

なくなっている。

例外処理が起きた場合は、①にある中間コード（補償コード）によりメモリへの書き込みを行い、ループの外では②にある中間コード（補償コード）でメモリへの書き込みを行うことにより、正しい動作をする。さらに、例外処理が起きない場合においては、メモリへの書き込み命令がループ内に存在しないため、図 1 6 と比べて高速に実行できる。

図 1 8 の例では、C a t c h 節のベーシックブロックに throw 以外の中間コードが入ったため、図 1 1 のステップ 1 1 0 5 における Try-Catch ブロックの取り外しは行われぬ。しかしながら、仮に C a t c h 節のベーシックブロック内に goto 命令または throw 命令だけの中間コードしか残らないならば、図 1 9 に示すようにして、Try-Catch ブロックを取り外す。

【 0 0 6 0 】

次に、本実施の形態を用いて例外処理に対応するための補償コードを挿入するさらに他の実施例として、関数呼び出しに関するメモリ書き込み命令の最適化について説明する。

従来、関数呼び出しを跨いで二つのメモリ書き込み命令の最適化を行う際には次の三つの条件が保証されていることが必要であった。

1. 関数の中で最適化対象のメモリへの書き込みがない。
2. 関数の中で最適化対象のメモリへの読み込みがない。
3. 関数の中で例外処理が起これない。

特に 3 の条件については、関数内で例外処理が起きる可能性があるとは判断されると、全てのメモリについての最適化が関数で KILL として扱っていたため止まっていた。そのため、ほとんどの関数呼び出しを超えて最適化を行うことはできなかった。そこで、本実施の形態における補償コードの挿入により関数呼び出しのコントロールフローグラフを変形すると、例外処理が発生する場合について当該例外処理に対応するための動作を行うこととなるため、上記 3 番目の「関数の中で例外処理が起これない」という条件を考慮する必要がなくなる。したがって、上記 1 番目及び 2 番目の条件が満たされていれば最適化が行えることとなる。

また、1 番目及び 2 番目の条件を満足しない場合であっても、全てのメモリに

付いての最適化が止まるわけではなく、保証できなかったメモリのみデータフローアナリシス上でKILLとして扱えば良い。したがって、最適化の性能向上が期待できる。

【 0 0 6 1 】

図 2 0 は、図 1 4 のサンプルプログラムにおける配列アクセス部分を関数に置き換えたプログラムを示す図である。また、図 2 1 は、図 2 0 において、関数 func 内で mem_pos に対する読み書きがないと分かった場合に、本実施の形態により補償コードを挿入してメモリの書き込み命令の最適化を行った状態を示す図である。

図 2 1 において、斜体の文字で書いた部分 2 1 0 1、2 1 0 2、2 1 0 3 が、例外処理が起きない場合にループ内で実行される部分である。図 2 0 と図 2 1 とを比較すると、ループ内において mem_pos への書き込みがなくなっていることがわかる。

【 0 0 6 2 】

〔第 2 の実施の形態〕

次に、本発明における他の実施の形態について説明する。

本発明の第 2 の実施の形態は、図 2 に示した第 1 の実施の形態における最適化処理部 1 0 と同様に構成され、前処理部 1 1、最適化処理実行部 1 2、後処理部 1 3 を備える。

【 0 0 6 3 】

本実施の形態において、前処理部 1 1 は、プログラム中の例外処理が起きる可能性のある命令を、ベーシックブロックとして、例外処理が起きるかどうかをチェックする命令と、実際に例外処理を起こす命令の二つに分ける。そして、これらの命令をコントロールフローグラフ上で明示的に分割する。そして、例外処理が起きる場合は、実際に例外処理を起こす命令に制御を移すようにする。

【 0 0 6 4 】

最適化処理実行部 1 2 は、通常の実最適化処理を実行する。最適化処理の対象であるプログラムは、前処理部 1 1 により例外処理が起きるかどうかをチェックする命令と、実際に例外処理を起こす命令とに分けられている。このため、例外処

理を起こさない場合には、当該例外処理を考慮せず（当該例外処理を起こす命令のパスを通らずに）処理を行うこととなる。したがって、効果の高い最適化を実現することができる。

【 0 0 6 5 】

後処理部 1 3 は、実際に例外処理を起こす命令があるベーシックブロックに、補償コードが生成されているかどうかをチェックする。そして、補償コードが生成されていない場合は、二つに分けた命令及びコントロールフローグラフを合成し、元の状態に戻す。

【 0 0 6 6 】

本実施の形態を実施するためには条件があるため、これについてまず説明する。例外処理が起きるかどうかをチェックするエクセプションチェック中間コードを実装する際には大きく分けて次の 2 つの実装方法がある。

第 1 は、ハードウェアの助けを借りて実装する方法である。第 2 は、明示的に例外処理をテストする命令を出し、条件分岐で例外処理が実際に起きる場合と起きない場合とで制御を分ける方法である。

【 0 0 6 7 】

第 1 のハードウェアを用いて実装する場合、配列境界チェックの実現を行っているならば、本実施の形態を適用すると、例外処理が起きない場合は、かえって処理が遅くなってしまう。したがって、本実施の形態では、第 2 の明示的に例外処理をテストする命令を出して、エクセプションチェック中間コードを実装することとする。

【 0 0 6 8 】

図 2 2 は、本実施の形態における最適化処理の概略的な流れを説明するフローチャートである。

図 2 2 を参照すると、オブジェクトコード（中間コード）に変換されたプログラムを入力した最適化処理部 1 0 において、まず、前処理部 1 1 が、メソッド内の例外処理が起きる可能性のある中間コードを例外処理が起きるかどうかをチェックする中間コードと、実際に例外処理を起こす中間コードとに分ける（ステップ 2 2 0 1）。そして、これらの中間コードを、コントロールフローグラフ上で

明示的に分割する（ステップ 2 2 0 2）。さらに、例外処理が起きる場合は、実際に例外処理を起こす中間コードに制御を移すようにコントロールフローグラフを構成する（ステップ 2 2 0 3）。

次に、最適化処理実行部 1 2 が、最適化処理を実行する（ステップ 2 2 0 4）。

次に、後処理部 1 3 が、前処理部 1 1 において分割された実際に例外処理を起こす中間コードがあるベーシックブロックに補償コードが生成されているかどうかをチェックする（ステップ 2 2 0 5）。そして、生成されていない場合は、前処理部 1 1 において分割された中間コード及びコントロールフローグラフを合成し、分割前の状態に戻す（ステップ 2 2 0 6）。

【 0 0 6 9 】

本実施の形態によれば、プログラムの実行時に例外処理が起きない場合、例外処理が起きる場合共に、本実施の形態を適用しない場合と比べて同等以上の実行速度が得られる。ただし、適用できる例外処理を起こす命令（エクセプションチェック中間コード）には制限がある。

そこで、本実施の形態は、上述した第 1 の実施の形態と組み合わせて用いることができる。具体的には、例えば次のような手順で、第 1、第 2 の実施の形態を組み合わせて適用することができる。

1. プログラム中の変形対象であるエクセプションチェック中間コードが、オブジェクトコードレベルで例外処理が起きるかどうかをチェックする部分とエクセプションを実際に起こす部分とで構成されている場合は、本実施の形態を使って変形する。
2. その他のエクセプションチェック中間コードに対しては、第 1 の実施の形態（Try-Catch ブロックを用いる方法）を使って変形する。
3. 通常の最適化を行う。
4. 最適化の対象とならなかった変形部分を元の中間コードに戻す。

【 0 0 7 0 】

次に、本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例として、第 1 の実施の形態においても説明した、部分的な冗長度除去（Pa

rtial Redundancy Elimination) のアルゴリズムを使ったメモリへの書き込み命令の最適化の効果を高める方法について説明する。

【 0 0 7 1 】

図 2 3 は、本実施例における動作を説明するフローチャートである。

図 2 3 を参照すると、まず、例外処理が起きる可能性のある中間コードを、例外処理が起きるかどうかをチェックする中間コードと、実際に例外処理を起こす中間コードとに分ける（ステップ 2 3 0 1）。

次に、例外処理が起きるかどうかをチェックする中間コードの下でベーシックブロックを分割し、実際に例外処理を起こす中間コードがあるベーシックブロックと、例外処理が起きる可能性のある中間コードに後続する中間コードの二つのコードに、コントロールフローグラフのエッジを張る（ステップ 2 3 0 2）。例外処理が起きる場合は、実際に例外処理を起こす中間コードに制御を移す。

【 0 0 7 2 】

次に、ステップ 2 3 0 2 で作成されたコントロールフローグラフに対して、メモリへの書き込み命令の最適化を行う（ステップ 2 3 0 3）。

最後に、ステップ 2 3 0 1 で作成した実際に例外処理を起こす中間コードがあるベーシックブロックに、例外処理を起こす中間コードしか存在しないならば、例外処理が起きるかどうかをチェックする中間コードと、実際に例外処理を起こす中間コードとを合成し、ステップ 2 3 0 1 における分割前の状態に戻す（ステップ 2 3 0 4）。

【 0 0 7 3 】

図 2 4 は例外処理が起きる可能性のある中間コードを含むプログラムのコントロールフローグラフ、図 2 5 は図 2 4 のコントロールフローグラフに対して図 2 3 に示した動作により補償コードを挿入し分岐を形成した状態のコントロールフローグラフを、それぞれ示す図である。

【 0 0 7 4 】

次に、本実施の形態におけるメモリへの書き込み命令の最適化の具体的な例を示す。

図 2 6 は、図 1 4 のサンプルプログラムに対して従来の最適化処理を実施して

得られた図 1 6 に示す状態のプログラムを対象として、本実施の形態による補償コードの挿入を行った状態を示す図である。

図 2 6 において、ループ内における例外処理が起きる可能性のある中間コードを、例外処理が起きるかどうかをチェックする中間コード `SIZECHECK_FAIL` と、実際に例外処理を起こす中間コード `SIZECHECK_EXCEPTION` の二つに分けている（2 6 0 1、2 6 0 2 参照）。`SIZECHECK_FAIL` は、例外処理が起きる場合に `TRUE` を返す中間コードを表している。

【0 0 7 5】

図 2 7 は、図 2 6 の状態のプログラムに対して、図 2 3 に示した部分的な冗長度除去（Partial Redundancy Elimination）のアルゴリズムを用いてメモリ書き込み命令の最適化を行った状態を示す図である。

図 2 7 において、斜体の文字で書いてある部分 2 7 0 1、2 7 0 2、2 7 0 3 は、例外処理が起きないときにループ内で実行される中間コードを表している。エクセプションが起きない場合のパスを考えると、図 1 6 と比べてメモリへの書き込みがなくなっている。

例外処理が起きた場合は、①にある中間コード（補償コード）によりメモリへの書き込みを行い、ループの外では②にある中間コード（補償コード）でメモリに書き込みを行うことにより、正しい動作をする。さらに、例外処理が起きない場合においては、メモリへの書き込み命令がループ内に存在しないため、図 1 6 と比べて高速に実行できる。

図 2 7 の例では、例外処理を起こす中間コードがあるベーシックブロックに例外処理を起こす中間コード以外の中間コードが入ったため、図 2 3 のステップ 2 3 0 4 における中間コードを合成し分割前の状態に戻す処理は行われず。しかしながら、仮に例外処理を起こす中間コードがあるベーシックブロックに例外処理を起こす中間コードしか残らないならば、図 2 8 に示すようにして、分割前の状態に戻す。

【0 0 7 6】

【発明の効果】

以上説明したように、本発明によれば、例外処理が起きる可能性のある命令に

対し、例外発生時、補償コードを経た後に当該例外処理を実行するようにプログラムを変形することにより、効果の高い最適化を行うことができる。

【図面の簡単な説明】

【図 1】 本発明により、例外処理が起きる可能性のある命令において、エクセプション・ハンドラへ移行する様子を示す図である。

【図 2】 第 1 の実施の形態によるコンパイラにおける最適化処理部の構成を説明する図である。

【図 3】 本実施の形態における最適化処理の概略的な流れを説明するフローチャートである。

【図 4】 本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例を説明する図であり、ベーシックブロックにおける要素の集合を求めるためのアルゴリズムを示す図である。

【図 5】 本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例を説明する図であり、データフローアナリシスを行うための関係式を示す図である。

【図 6】 本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例を説明する図であり、ローカル変数の集合を求めるためのアルゴリズムを示す図である。

【図 7】 本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例を説明する図であり、所定の命令に対して新たなエクセプション・ハンドラを設定するためのアルゴリズムを示す図である。

【図 8】 図 7 のアルゴリズムにおける＜この命令について新たなエクセプション・ハンドラを設定する。＞の処理内容を説明するフローチャートである。

【図 9】 配列内の最小値及び最大値を求める J a v a のサンプルプログラムを示す図である。

【図 1 0】 図 9 のプログラムに対して本実施の形態により補償コードを挿入する変形を加えた状態を示す図である。

【図 1 1】 本実施の形態を用いて例外処理に対応するための補償コードを挿入する他の実施例として、部分的な冗長度除去のアルゴリズムを使ったメモリ

への書き込み命令の最適化の効果を高める方法を説明するフローチャートである。

【図 1 2】 例外処理が起きる可能性のある中間コードを含むプログラムのコントロールフローグラフを示す図である。

【図 1 3】 図 1 2 のコントロールフローグラフに対して補償コードを挿入した状態を示すコントロールフローグラフを示す図である。

【図 1 4】 J a v a のサンプルプログラムを示す図である。

【図 1 5】 図 1 4 のサンプルプログラムを細かい処理単位に分けたプログラムを示す図である。

【図 1 6】 図 1 5 に示すプログラムに対して、従来技術による最適化処理を行った状態を示す図である。

【図 1 7】 図 1 6 の状態のプログラムに対して、本実施の形態による補償コードの挿入を行った状態を示す図である。

【図 1 8】 図 1 7 の状態のプログラムに対し、部分的な冗長度除去のアルゴリズムを用いてメモリ書き込み命令の最適化を行った状態を示す図である。

【図 1 9】 図 1 8 の状態のプログラムから Try-Catch ブロックを取り外す様子を示す図である。

【図 2 0】 図 1 4 のサンプルプログラムにおける配列アクセス部分を関数に置き換えたプログラムを示す図である。

【図 2 1】 図 2 0 において、補償コードを挿入してメモリの書き込み命令の最適化を行った状態を示す図である。

【図 2 2】 第 2 の実施の形態における最適化処理の概略的な流れを説明するフローチャートである。

【図 2 3】 本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例として、部分的な冗長度除去のアルゴリズムを使ったメモリへの書き込み命令の最適化の効果を高める方法を説明するフローチャートである。

【図 2 4】 例外処理が起きる可能性のある中間コードを含むプログラムのコントロールフローグラフを示す図である。

【図 2 5】 図 2 4 のコントロールフローグラフに対して補償コードを挿入

し分岐を形成した状態のコントロールフローグラフを示す図である。

【図 2 6】 図 1 6 に示す状態のプログラムを対象として、本実施の形態による補償コードの挿入を行った状態を示す図である。

【図 2 7】 図 2 6 の状態のプログラムに対して、部分的な冗長度除去のアルゴリズムを用いてメモリ書き込み命令の最適化を行った状態を示す図である。

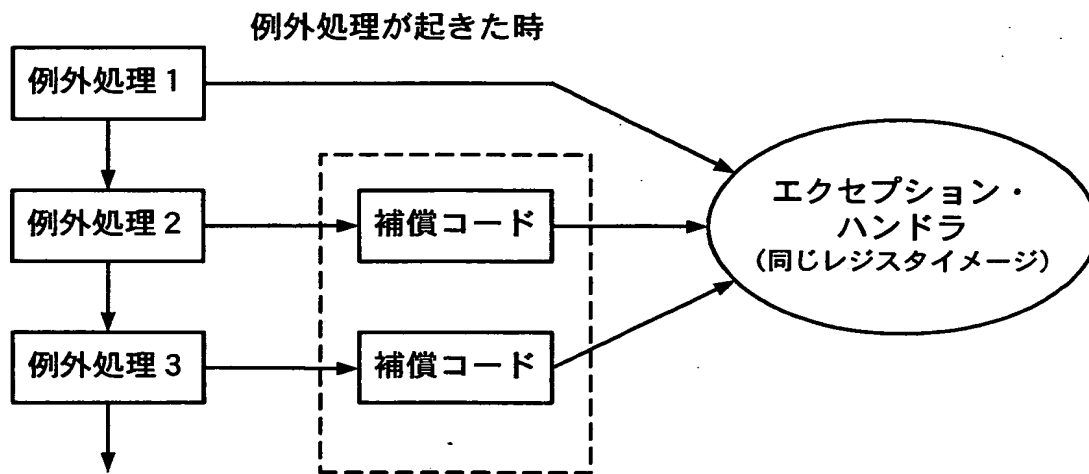
【図 2 8】 図 2 7 の状態のプログラムから不要な分岐を分割前の状態に戻す様子を示す図である。

【符号の説明】

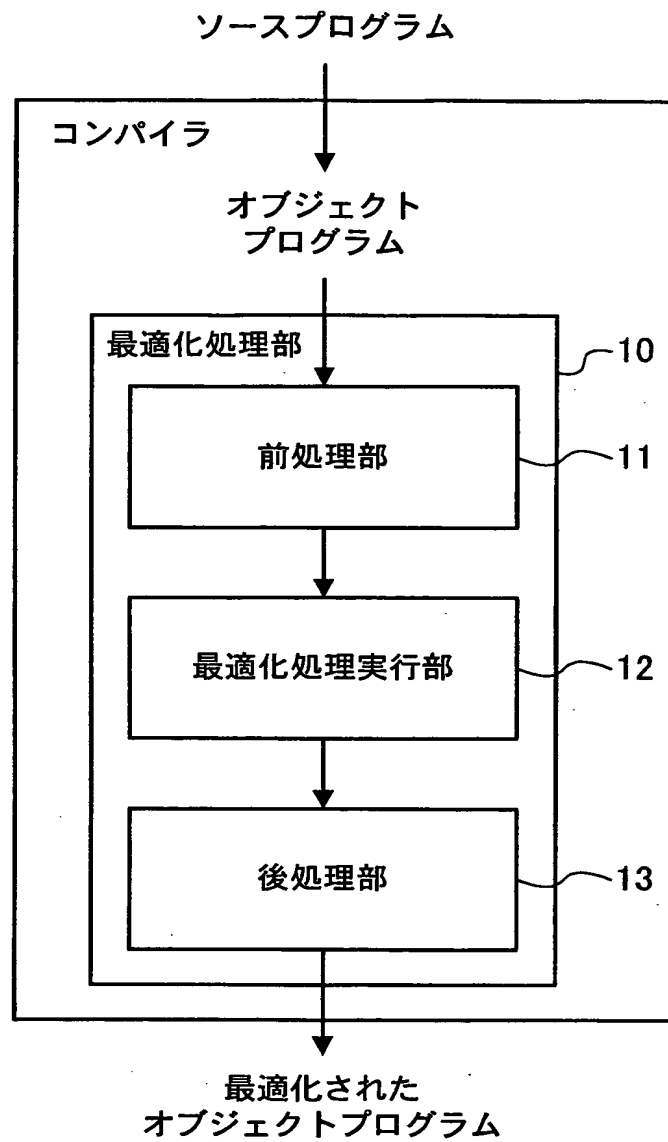
1 0 …最適化処理部、 1 1 …前処理部、 1 2 …最適化処理実行部、 1 3 …後処理部

【書類名】 図面

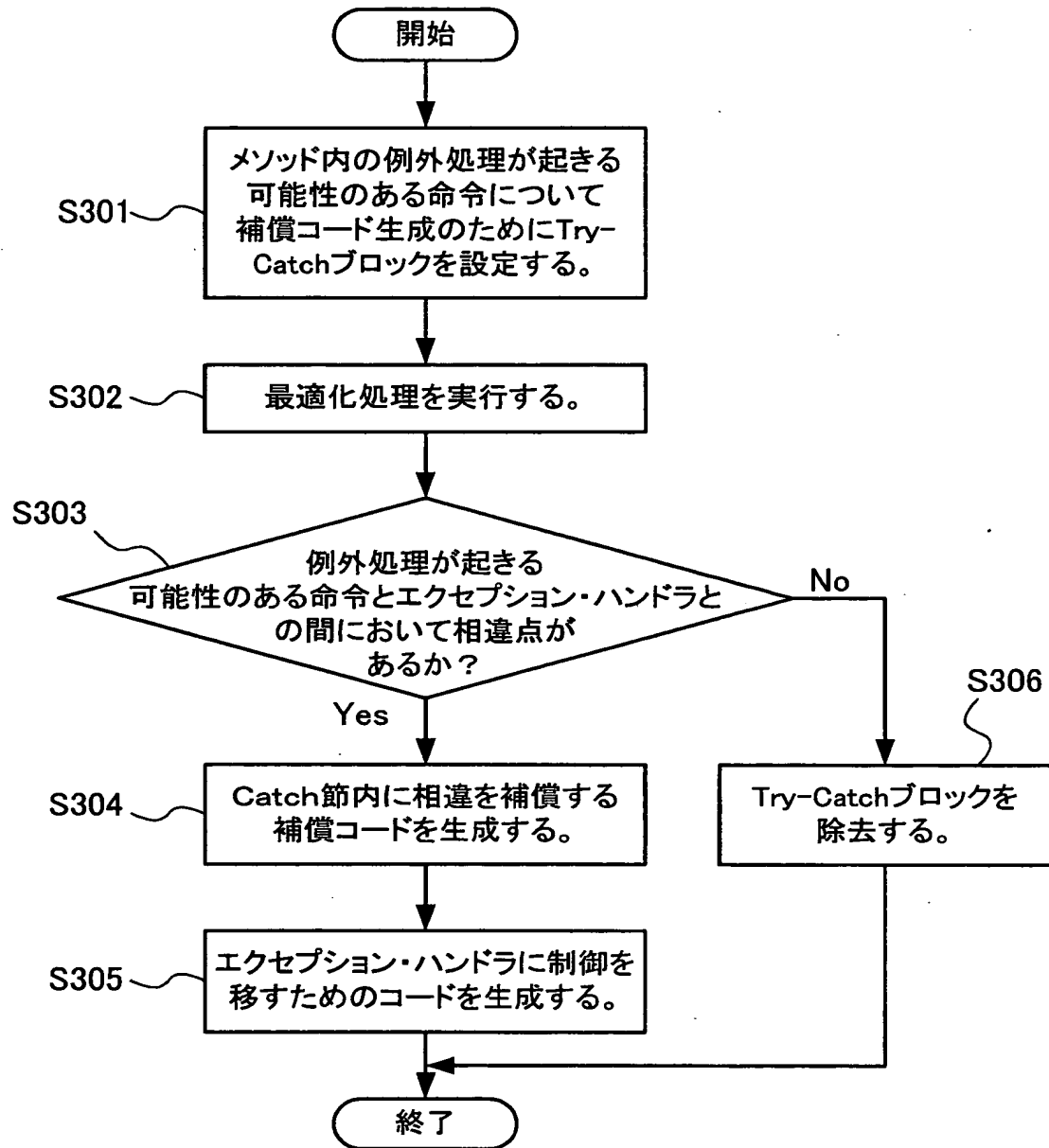
【図 1】



【図 2】



【図 3】



【図 4】

```

for (すべてのbasic blockについて繰り返す) {
  bidx = basic blockのインデックス
  if (このbasic blockがエクセプション・ハンドラの開始点) {
    reach[bidx].gen = ( このエクセプション・ハンドラに対応する全てのTRY,
                        全てのローカル変数 );
  } else {
    reach[bidx].gen =  $\phi$ 
  }
  reach[bidx].kill =  $\phi$ 
  for (basic block内のすべての命令について実行順に繰り返す) {
    if (命令 == ローカル変数へのwrite) {
      reach[bidx].kill  $\cup$  = ( 全てのTRY, 書き込まれたローカル変数番号 );
    }
  }
}

```

【図 5】

$$\text{reach}[B].\text{in} = \left(\bigcup_{P \in \text{Pred}(B)} \text{reach}[P].\text{out} \right) \cup \text{reach}[B].\text{gen}$$

$$\text{reach}[B].\text{out} = \text{reach}[B].\text{in} - \text{reach}[B].\text{kill}$$

【図 6】

```

for (全てのTRYについて繰り返す) USE_VIA_EH_LIST[TRY] = φ;
for (すべてのbasic blockについて繰り返す) {
    bidx = basic blockのインデックス;
    if (reach[bidx].in != φ) {
        for (basic block内のすべての命令について実行順に繰り返す) {
            switch (命令) {
                case ローカル変数へのwrite:
                    reach[bidx].in -= ( 全てのTRY, 書き込まれたローカル変数番号 );
                    break;
                case ローカル変数からのread:
                    for (全てのTRYについて繰り返す) {
                        if ( (TRY, 読み込まれたローカル変数番号) ∈ reach[bidx].in ) {
                            USE_VIA_EH_LIST[TRY] ∪= 読み込まれたローカル変数番号;
                        }
                    }
                    break;
            }
        }
    }
}

```

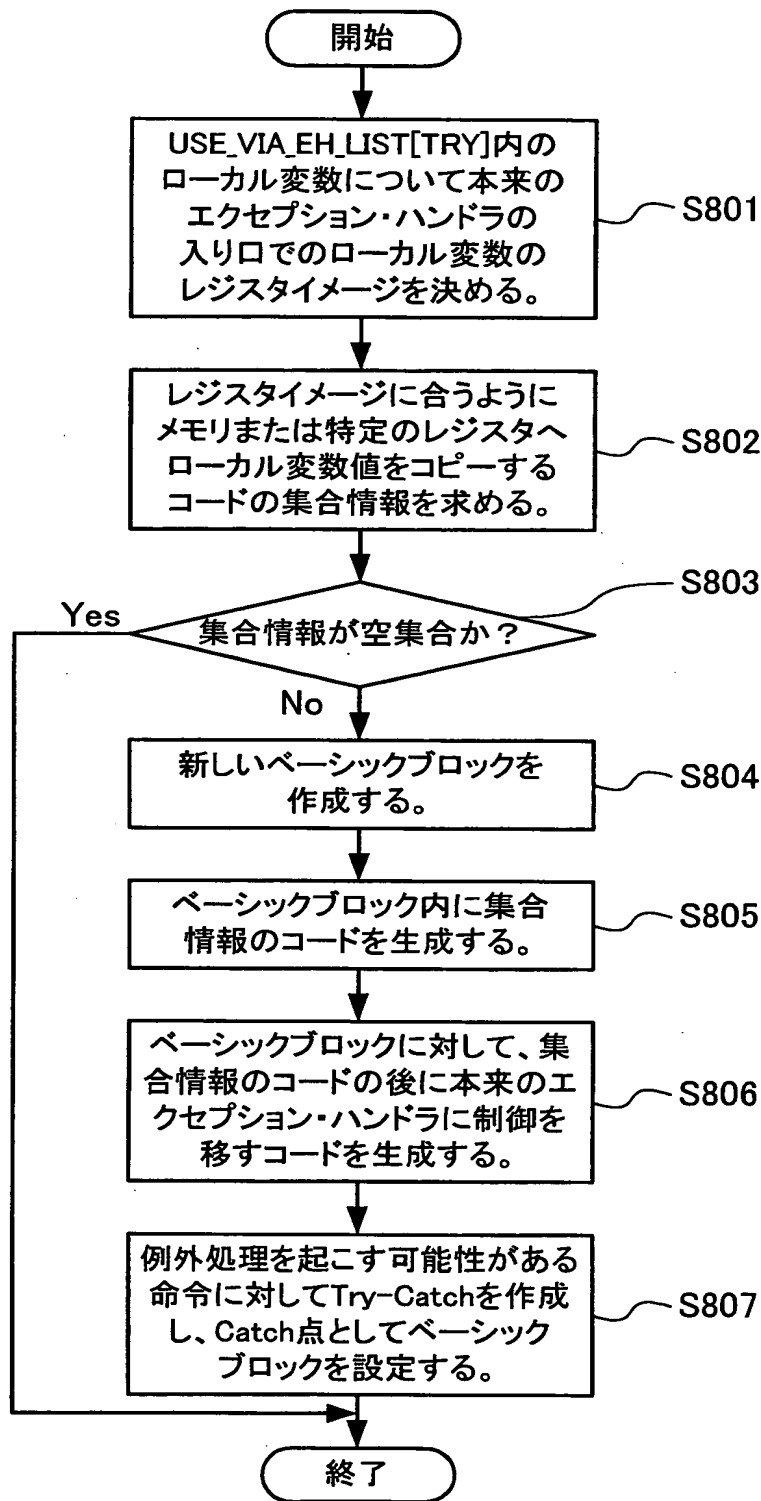
【図 7】

```

for (すべてのbasic blockについて繰り返す) {
    if (basic blockがtry regionに含まれる) {
        TRY = Try Region の識別番号
        if (USE_VIA_EH_LIST[TRY] != φ) {
            bidx = basic blockのインデックス;
            for (basic block内のすべての命令について実行順に繰り返す) {
                if (命令 ∈ エクセプションを起こす可能性がある命令) {
                    if (起きる可能性があるエクセプションに対応するエクセプション・ハンドラがある) {
                        < この命令について新たなエクセプション・ハンドラを設定する。 >
                    }
                }
            }
        }
    }
}

```

【図 8】



【図 9】

```

int MIN_VAL, MAX_VAL;
void sample(int a[], int size_x, int size_y) {
    int min, max;
    int i, j;
    i = 0;          --- (1)
    j = 0;          --- (2)
    try {
        min = a[i][j]; --- (3)
        max = min;    --- (4)
        i = 0;        --- (5)
        while(i < size_x) { --- (6)
            j = 0;    --- (7)
            while(j < size_y) { --- (8)
                int val = a[i][j]; --- (9)
                if (min > val) { --- (10)
                    min = val; --- (11)
                }
                if (max < val) { --- (12)
                    max = val; --- (13)
                }
                j++; --- (14)
            }
            i++; --- (15)
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("ArrayIndexOutOfBoundsException i=" + i + " j=" + j); --- (16)
        /* error status : min > max */
        max = 0x80000000; --- (17)
        min = 0x7FFFFFFF; --- (18)
    }
    MIN_VAL = min; --- (19)
    MAX_VAL = max; --- (20)
}

```

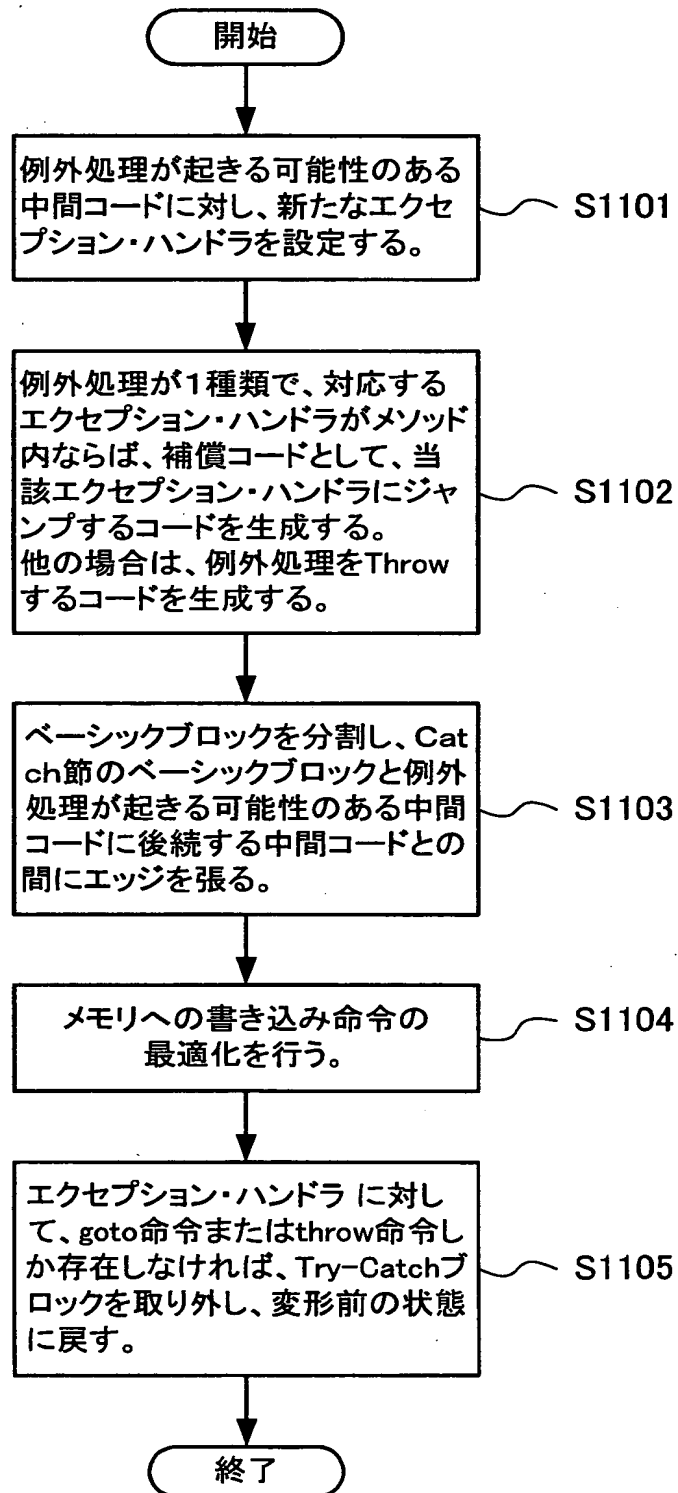
【図 10】

```

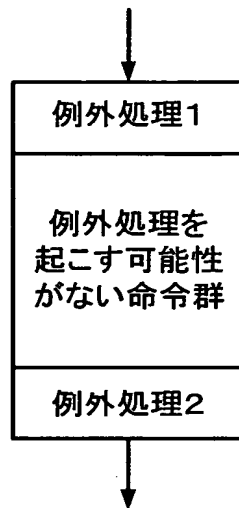
int MIN_VAL, MAX_VAL;
void sample(int a[], int size_x, int size_y) {
    int min, max;
    int i, j;
    i = 0;          --- (1)
    j = 0;          --- (2)
    try {
        NULLCHECK a;          --- (3.1)
        try {
            SIZECHECK i, a[];  --- (3.2)
        } catch (ArrayIndexOutOfBoundsException e) {
            i と j の変数値をそれぞれR1, R2にコピーする。
            goto Handler;
        }
        try {
            SIZECHECK j, a[i];  --- (3.3)
        } catch (ArrayIndexOutOfBoundsException e) {
            i と j の変数値をそれぞれR1, R2にコピーする。
            goto Handler;
        }
        min = a[i][j];          --- (3.4)
        max = min;              --- (4)
        i = 0;                  --- (5)
        while(i < size_x) {      --- (6)
            j = 0;              --- (7)
            while(j < size_y) {  --- (8)
                try {
                    SIZECHECK i, a[]; --- (9.1)
                } catch (ArrayIndexOutOfBoundsException e) {
                    i と j の変数値をそれぞれR1, R2にコピーする。
                    goto Handler;
                }
                try {
                    SIZECHECK j, a[i]; --- (9.2)
                } catch (ArrayIndexOutOfBoundsException e) {
                    i と j の変数値をそれぞれR1, R2にコピーする。
                    goto Handler;
                }
                int val = a[i][j]; --- (9.3)
                if (min > val) {    --- (10)
                    min = val;    --- (11)
                }
                if (max < val) {    --- (12)
                    max = val;    --- (13)
                }
                j++;              --- (14)
            }
            i++;                  --- (15)
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        Handler:
        /* i, jの値はそれぞれR1, R2に入っている。 */
        System.err.println("ArrayIndexOutOfBounds i=" + i + " j=" + j); --- (16)
        /* error status : min > max */
        max = 0x80000000; --- (17)
        min = 0x7FFFFFFF; --- (18)
    }
    MIN_VAL = min;              --- (19)
    MAX_VAL = max;              --- (20)
}

```

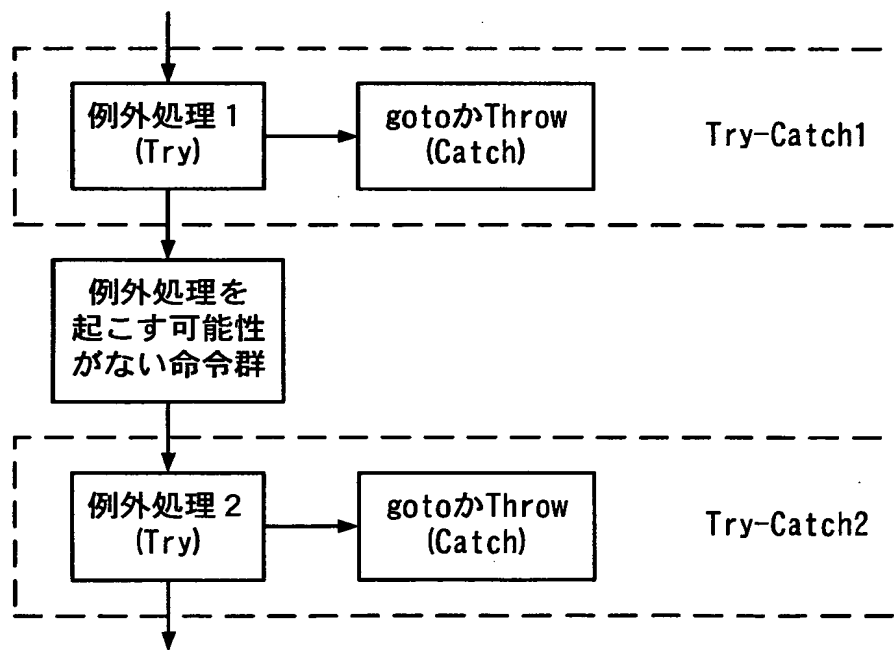
【図 1 1】



【図 1 2】



【図 1 3】



【図 1 4】

```

static int mem_pos;      // メモリ上にある変数
static int mem_a[];      // メモリ上にある変数
void SearchPos(int reg_data)
{
    mem_pos = 0;
    while (mem_a[mem_pos] != reg_data) {
        mem_pos++;
    }
}

```

【図 1 5】

```

static int mem_pos;      // メモリ上にある変数
static int mem_a[];      // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;          // レジスター上にある変数
    int reg_a[];          // レジスター上にある変数

    mem_pos = 0;

    goto entry;
loop:
    reg_pos = mem_pos;
    reg_pos++;
    mem_pos = reg_pos;
entry:
    reg_pos = mem_pos;
    reg_a = mem_a;
    NULLCHECK reg_a;      // 例外処理が起きる可能性がある中間コード
    SIZECHECK reg_pos, reg_a[]; // 例外処理が起きる可能性がある中間コード
    if (reg_a[reg_pos] != reg_data) goto loop;
}

```


【図 16】

```
static int mem_pos;    // メモリ上にある変数
static int mem_a[];    // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;        // レジスター上にある変数
    int reg_a[];        // レジスター上にある変数

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    NULLCHECK reg_a;    // 例外処理が起きる可能性がある中間コード

    goto entry;
loop:
    reg_pos++;
    mem_pos = reg_pos; -① // この中間コードをループ内から出すことができない
entry:
    SIZECHECK reg_pos, reg_a[]; // 例外処理が起きる可能性がある中間コード
    if (reg_a[reg_pos] != reg_data) goto loop;
}
```

1601

1602

【図 17】

```
static int mem_pos;    // メモリ上にある変数
static int mem_a[];    // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;        // レジスター上にある変数
    int reg_a[];        // レジスター上にある変数

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    try {
        NULLCHECK reg_a; // 例外処理が起きる可能性がある中間コード
    } catch (Throwable t) {
        throw t;
    }
    goto entry;
loop:
    reg_pos++;
    mem_pos = reg_pos;
entry:
    try {
        SIZECHECK reg_pos, reg_a[]; // 例外処理が起きる可能性がある
                                     // 中間コード
    } catch (Throwable t) {
        throw t;
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
}
```

1701

1702

【図 18】

```

static int mem_pos;      // メモリ上にある変数
static int mem_a[];      // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;          // レジスター上にある変数
    int reg_a[];          // レジスター上にある変数

    reg_pos = 0;
    reg_a = mem_a;
    try {
        NULLCHECK reg_a; // 例外処理が起きる可能性がある中間コード
    } catch (Throwable t) {
        mem_pos = reg_pos;
        throw t;
    }

    goto entry;           1801
loop:
    reg_pos++;
entry:
    try {
        SIZECHECK reg_pos, reg_a[]; // 例外処理が起きる可能性がある
                                     // 中間コード
    } catch (Throwable t) {
        mem_pos = reg_pos;          ①
        throw t;
    }

    if (reg_a[reg_pos] != reg_data) goto loop;
    mem_pos = reg_pos;             ②           1803
}

```

【図 19】

```

try {
    ExceptionCheck; // 例外処理が起きる可能性がある中間コード
} catch (Throwable t) {
    throw t;
}

```



```

ExceptionCheck; // 例外処理が起きる可能性がある中間コード

```

【図 2 0】

```

static int mem_pos;    // メモリ上にある変数
void SearchPos(int reg_data)
{
    mem_pos = 0;
    while (func(mem_pos) != reg_data) {
        mem_pos++;
    }
}

```

【図 2 1】

```

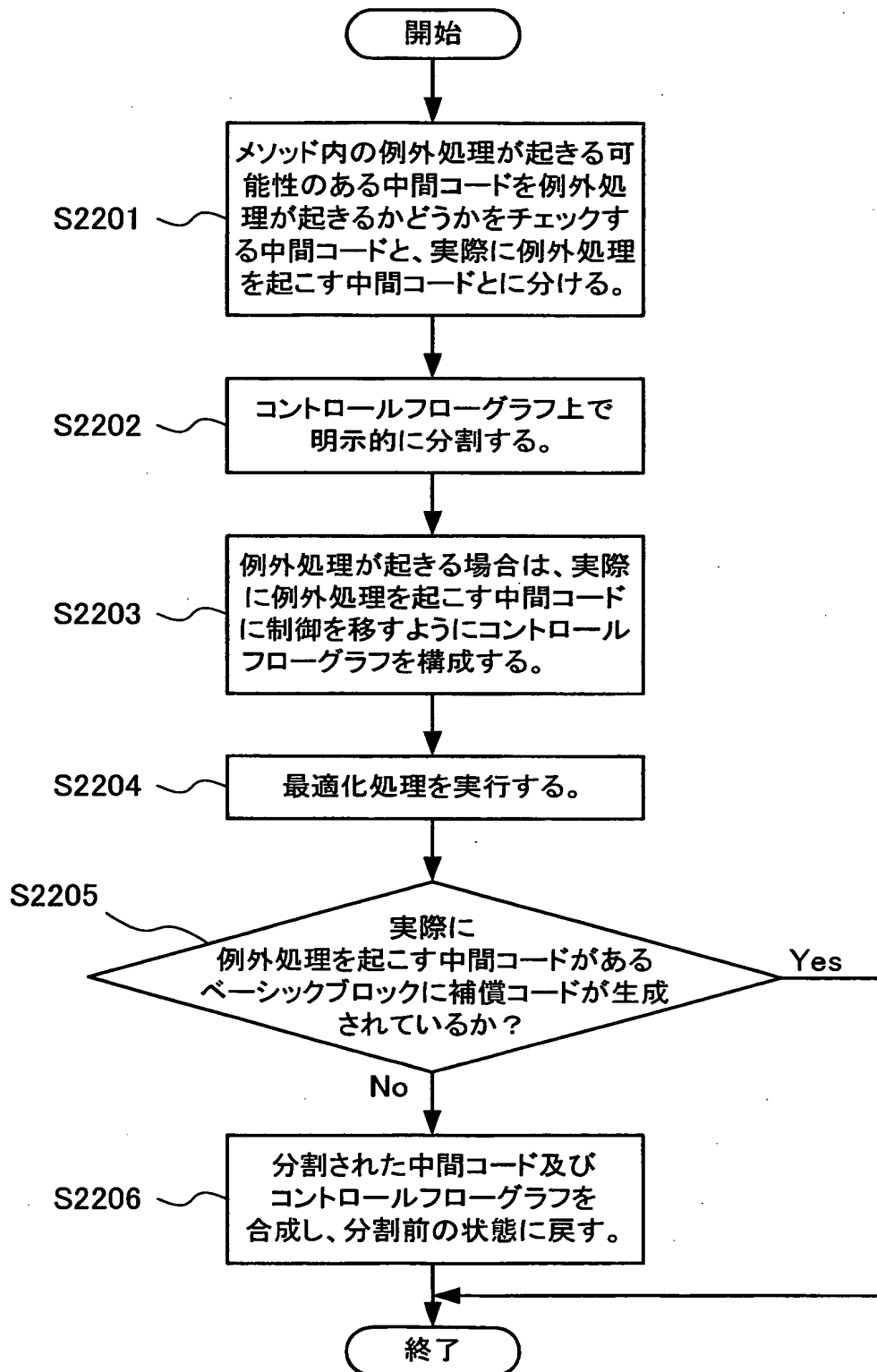
static int mem_pos;    // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos; // レジスター上にある変数
    int reg_ret; // レジスター上にある変数

    reg_pos = 0;

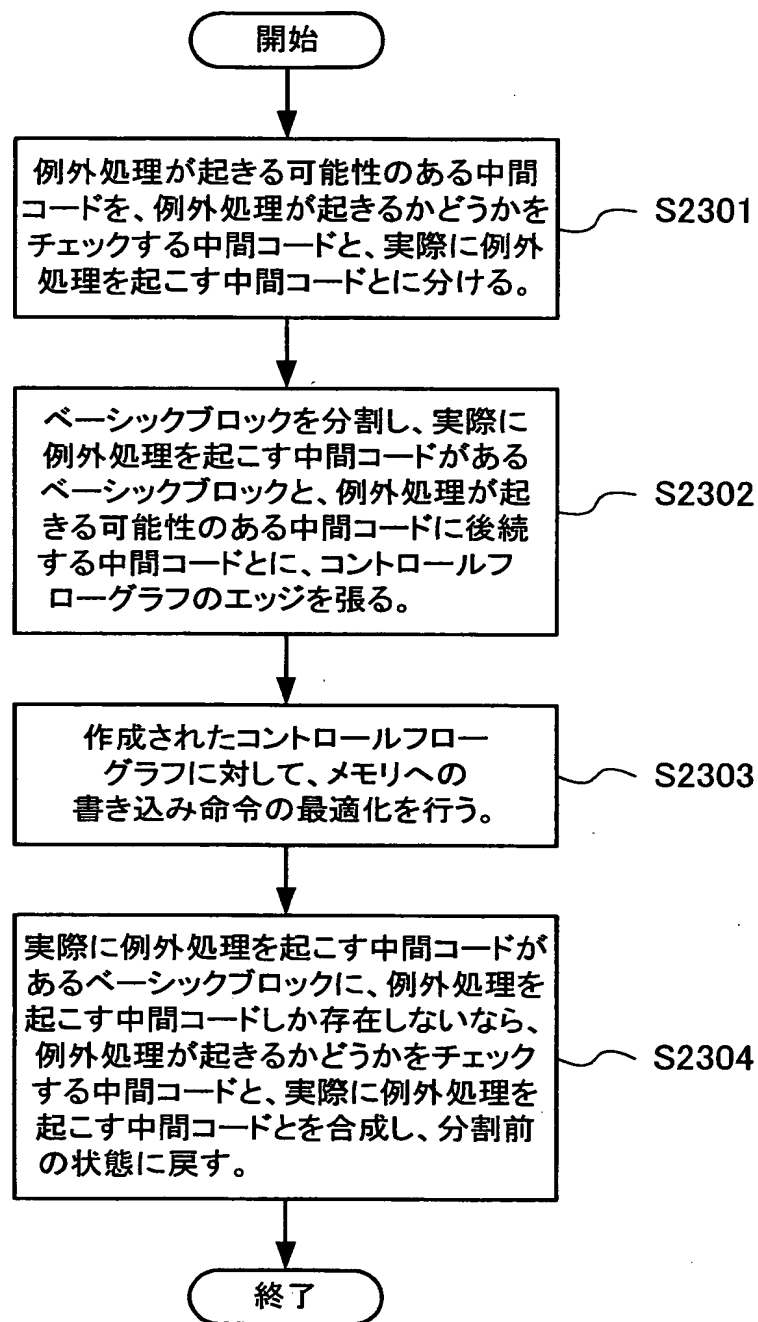
    goto entry;
loop:
    reg_pos++; ▲ 2101
entry:
    try {
        reg_ret = func(reg_pos); ▲ 2102
    } catch (Throwable t) {
        mem_pos = reg_pos;
        throw t;
    }
    if (reg_ret != reg_data) goto loop; ▲ 2103
    mem_pos = reg_pos;
}

```

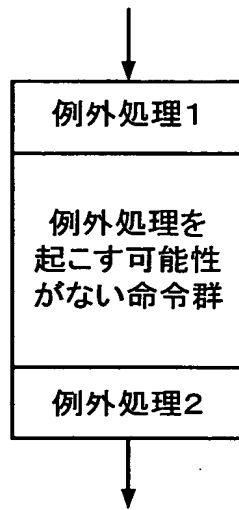
【図 2 2】



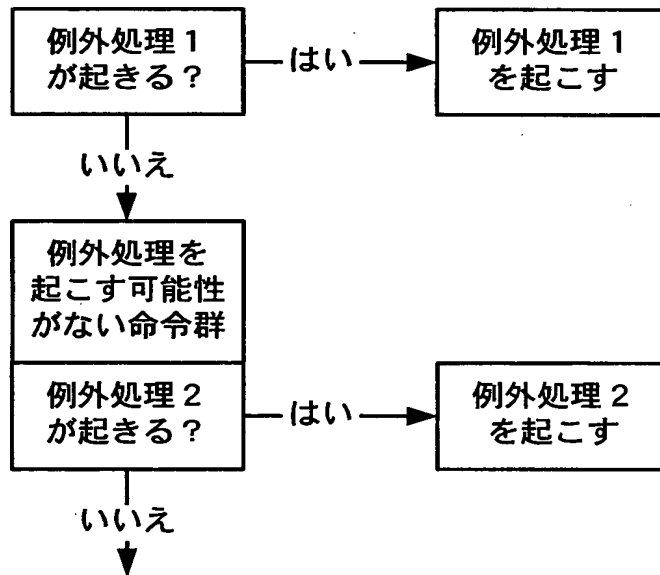
【図 23】



【図 2 4】



【図 2 5】



【図 2 6】

```

static int mem_pos;      // メモリ上にある変数
static int mem_a[];      // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;          // レジスター上にある変数
    int reg_a[];          // レジスター上にある変数

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    if (NULLCHECK_FAIL reg_a) { // 例外処理が起きるかどうか
                                // チェックする中間コード
        NULLCHECK_EXCEPTION reg_a; // 例外処理を起こす中間コード
    }

    goto entry;
loop:
    reg_pos++;
    mem_pos = reg_pos;
entry:
    if (SIZECHECK_FAIL reg_pos, reg_a[]) { // 例外処理が起きるかどうか
                                            // チェックする中間コード
        SIZECHECK_EXCEPTION reg_pos, reg_a[]; // 例外処理を起こす中間コード
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
}

```

2601

2602

【図 2 7】

```

static int mem_pos;      // メモリ上にある変数
static int mem_a[];      // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;          // レジスター上にある変数
    int reg_a[];          // レジスター上にある変数

    reg_pos = 0;
    reg_a = mem_a;
    if (NULLCHECK_FAIL reg_a) { // 例外処理が起きるかどうか
                                // チェックする中間コード
        mem_pos = reg_pos;
        NULLCHECK_EXCEPTION reg_a; // 例外処理を起こす中間コード
    }

    goto entry;
loop:
    reg_pos++;             ← 2701
entry:
    if (SIZECHECK_FAIL reg_pos, reg_a[]) { // 例外処理が起きるかどうか
                                            // チェックする中間コード
        mem_pos = reg_pos;             -①
        SIZECHECK_EXCEPTION reg_pos, reg_a[]; // 例外処理を起こす中間コード
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
    mem_pos = reg_pos;                 -②
}

```

2702

2703

【図 2 8】

```

if (EXCEPTIONCHECK_FAIL) { // 例外処理が起きるかどうか
                           // チェックする中間コード
    EXCEPTION;             // 例外処理を起こす中間コード
}

```



```

ExceptionCheck; // 例外処理が起きる可能性がある中間コード

```

【書類名】 要約書

【要約】

【課題】 例外処理が起きる可能性のある命令を含むプログラムに対して効果の高い最適化を行う。

【解決手段】 プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラにおいて、機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部 1 2 と、このオブジェクトプログラムに対して、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点とこの例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行う前処理部 1 1 及び後処理部 1 3 とを備える。

【選択図】 図 2

認定・付加情報

特許出願の番号	特願 2000-114193
受付番号	50000477251
書類名	特許願
担当官	佐藤 一博 1909
作成日	平成 12 年 5 月 31 日

<認定情報・付加情報>

【特許出願人】

【識別番号】	390009531
【住所又は居所】	アメリカ合衆国 10504、ニューヨーク州 アーモンク (番地なし)
【氏名又は名称】	インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】

【識別番号】	100086243
【住所又は居所】	神奈川県大和市下鶴間 1623 番地 14 日本アイ・ビー・エム株式会社 大和事業所内
【氏名又は名称】	坂口 博

【復代理人】

【識別番号】	100104880
【住所又は居所】	東京都港区赤坂 7-10-9 第 4 文成ビル 202 セリオ国際特許事務所
【氏名又は名称】	古部 次郎

【選任した代理人】

【識別番号】	100091568
【住所又は居所】	神奈川県大和市下鶴間 1623 番地 14 日本アイ・ビー・エム株式会社 大和事業所内
【氏名又は名称】	市位 嘉宏

【選任した復代理人】

【識別番号】	100100077
【住所又は居所】	東京都港区赤坂 7-10-9 第 4 文成ビル 202 セリオ国際特許事務所
【氏名又は名称】	大場 充

出 願 人 履 歴 情 報

識別番号 [390009531]

1. 変更年月日 1990年10月24日
[変更理由] 新規登録
住 所 アメリカ合衆国10504、ニューヨーク州 アーモンク (番地なし)
氏 名 インターナショナル・ビジネス・マシーンズ・コーポレイション

2. 変更年月日 2000年 5月16日
[変更理由] 名称変更
住 所 アメリカ合衆国10504、ニューヨーク州 アーモンク (番地なし)
氏 名 インターナショナル・ビジネス・マシーンズ・コーポレイション